Test et Validation du Logiciel

Génération des données de test

1 Tests parametrisés

1.1 Format CSV

Depuis la version 5 de jUnit, il est possible de regrouper des cas de test en utilisant les annotations montrées sur l'exemple suivant :

```
@ParameterizedTest
@CsvSource({"0, 1, 0", "1, 0, 0", "1, 1, 1"})
void testProd(int x, int y, int expected) {
   int actual = calc.prod(x, y);
   assertEquals(expected, actual);
}
```

La syntaxe utilisée est le format CSV (*Comma Separated Values*), avec une chaîne de caractères par cas de test. Le nombre et l'ordre des arguments donnés doit correspondre à ceux de la fonction de test. Des heuristiques sont utilisées pour reconnaître les valeurs des arguments et les convertir vers le type attendu.

Les détails sont disponibles dans la documentation de jUnit 5 :

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

Question

1. Reprendre les exercices du TP précédent, et regrouper les différentes données de test de chaque fonction en utilisant le format CSV.

1.2 Fichier CSV externe

Créer un nouveau fichier `"testProd2.csv"` dans le répertoire `test`, et le remplir avec quelques données de test au format CSV. On pourra utiliser simple éditeur de texte, ou un tableur, pour saisir et éditer ces données de test, ou installer un plugin Eclipse adapté.

```
@ParameterizedTest
@CsvFileSource(resources = "/testProd2.csv")
void testProd2(int x, int y, int expected) {
   int actual = calc.prod(x, y);
   assertEquals(expected, actual);
}
```

Question

1. Ecrire une fonction qui prend en entrée trois `String` et renvoie la plus courte. Tester cette fonction en regroupant les données de test dans un fichier CSV externe. On fera attention aux taux de couverture de ces tests.

1.3 Données de test « générées »

Il est aussi possible générer les cas de test, et de les stocker dans une collection, ou sous forme de `stream`.

```
static IntStream range() {
   return IntStream.range(0, 20).skip(10);
}

@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
   int actual = calc.add(x, x);
   int expected = 2 * x;
   assertEquals(expected, actual);
}
```

Ces données de test peuvent même être aléatoires comme, par exemple, 20 nombres entiers aléatoires n tels que $0 \leqslant n < 100$:

```
static IntStream random() {
   return new Random().ints(0, 100).limit(20);
}

@ParameterizedTest
@MethodSource("random")
void testadd3(int x) {
   int actual = calc.add(x, x);
   int expected = 2 * x;
   assertEquals(expected, actual);
}
```

Question

1. Tester l'exemple ci-dessus, puis modifier l'annotation `@ParameterizedTest` de manière à pour afficher les valeurs aléatoires qui sont testées :

   ```
   @ParameterizedTest(name = "Trying {0}")
   ```

Remarques. Ce type de tests aléatoires apporte toutefois de nouvelles difficultés :

– quel résultat attendre pour un test donné ?

– comment générer des données aléatoires pertinentes pour des types complexes ?

La réponse à la première question dépend des exigences. On se contente généralement de vérifier une propriété donnée du résultat (la *post-condition* du test). Ces propriétés peuvent caractériser la fonction testée complètement (par exemple, pour une fonction qui trie un tableau de valeurs, il suffit de vérifier que le résultat contient bien les mêmes valeurs dans l'ordre), ou seulement partiellement (par exemple, la même taille). On parle alors de *Property Based Testing*[1].

La réponse à la seconde question nécéssite d'une part de fournir des générateurs aléatoires pour des types de données structurés, et d'autre part de filtrer les entrées pertinentes des tests à l'aide d'une *pré-condition*.

1. La première implantation de *Property Based Testing*, appelée QuickCheck, a été développée pour Haskell, mais des implantations récentes existent aussi pour Java comme jqwick (https://jqwik.net) ou ScalaCheck (https://www.scalacheck.org).

Questions

1. Reprendre l'exercice précédent : supprimer le résultat attendu des données de test et définir la bonne post-condition.

2. Remplacer les données de test de la question précédente par des données aléatoires. Utiliser l'annotation `@RepeatedTest` pour faire répéter ce test 20 fois.

2 Exercices

2.1 Test d'une fonction de tri

Considérons la fonction suivante qui implante l'algorithme du tri à bulle :

```
static void sort(Integer[] num) {
   boolean flag = true;              // set flag to true to begin first pass
   while (flag) {
      flag = false;                  // set flag to false awaiting a possible swap
      for (int j = 0; j < num.length-1; j++) {
         if (num[j] > num[j+1]) {    // ascending sort
             int temp = num[j];      // swap elements
             num[j] = num[j+1];
             num[j+1] = temp;
             flag = true;            // a swap occurred
         }
      }
   }
}
```

Questions

1. *Tests structurels* : écrire les tests jUnit 5 pour cette fonction permettant d'obtenir la couverture des instructions, puis des décisions.

2. Proposer une post-condition qui exprime le fait que le tableau est bien trié en sortie. Ecrire une fonction booléenne `isSorted` qui permet de le vérifier.

3. *Tests fonctionnels* : compléter vos tests avec un test aléatoire. On définira une fonction auxiliaire `randomArray` qui crée un tableau de taille aléatoire inférieure ou égale à 10 contenant des valeurs aléatoire inférieures à 100. Ce test sera répété 20 fois.

2.2 Test d'une implantation des piles

La classe `Stack` ci-dessous implémente les primitives habituelles (`push`, `pop`, `isFull`, `isEmpty`) pour des piles d'entiers de taille bornée, dont les éléments sont stockés dans le tableau `elems` et où `top` correspond à la première case libre du tableau.

```
class Stack {

    Integer[] elems;
    int top;

    public Stack(int capacity) {
        elems = new Integer[capacity];
        top = 0;
    }
```

```
    @Override
    public String toString() {
        return Arrays.asList(elems).subList(0, top).toString();
    }

    public void push(Integer i) {
        elems[top++] = i;
    }

    public Integer pop() {
        return elems[--top];
    }

    public boolean isFull() {
        if (top == elems.length)
            return true;
        else
            return false;
    }

    public boolean isEmpty() {
        if (top == 0)
            return true;
        else
            return false;
    }
}
```

Les exigences qui correspondent à ce code sont les suivantes :

- la pile est correctement construite : `elems` est bien initialisés, et `top` est supérieur ou égal à 0 et inférieur `elems.length`.
- si on empile (`push`) un élément sur une pile supposée non pleine, la pile contient alors un élément de plus et son sommet est l'élément empilé.
- si on dépile (`pop`) un élément d'une pile supposée non vide, la pile contient alors un élément de moins et on retourne l'ancien sommet de la pile.
- `isFull` détermine bien si la pile est pleine.
- `isEmpty` détermine bien si la pile est vide.

Questions

1. Traduire ces exigences en propriétés de la forme *assume*/*assert*, et les programmer en utilisant jUnit 5. Chaque méthode de test prendra la pile à tester en paramètre.
2. *Tests structurels* : proposer des données de test pour ces primitives permettant d'obtenir la couverture des instructions, puis des décisions. On utilisera pour cela des tests paramétrés et une méthode auxiliaire `initStack` permettant de créer une pile déjà partiellement remplie (et de capacité égale à 16).
3. *Tests fonctionnels* : compléter vos tests avec des tests aléatoires. On utilisera cette fois une fonction auxiliaire `randomStack` qui crée une pile de hauteur aléatoire déjà partiellement remplie par des valeurs aléatoires (mais toujours de capacité fixe égale à 16). Une suite de 20 piles aléatoires sera générée.

2.3 Test d'une implantation des files d'attente

La classe `Queue` ci-dessous implémente les primitives habituelles (`enqueue`, `dequeue`, `isFull`, `isEmpty`) pour des files d'attente de taille bornée (par `capacity`), dont les éléments sont des entiers stockés dans le tableau `buffer` et où `size` correspond au nombre d'éléments dans la file d'attente, et `front` et `rear` correspondent respectivement à la tête et à la fin de la file.

```
class Queue
{
    int front;
    int rear;
    int size;
    Integer[] buffer;

    public Queue(int capacity)
    {
        front = 0;
        rear = 0;
        size = 0;
        buffer = new Integer[capacity];
    }

    public boolean isEmpty()
    {
        return (size == 0);
    }

    public boolean isFull()
    {
        return (size == buffer.length);
    }

    public void enqueue(Integer i)
    {
        size++;
        rear = (rear + 1) % buffer.length;
        buffer[rear] = i;
    }

    public Integer dequeue()
    {
        size--;
        front = (front + 1) % buffer.length;
        return buffer[front];
    }
}
```

Les exigences qui correspondent à ce code sont les suivantes :

- une file d'attente est correctement construite : `buffer` est bien initialisé et de taille `capacity`, et `size`, `front` et `rear` sont à 0.
- si on ajoute (`enqueue`) un élément dans une file d'attente supposée non pleine, la file contient alors un élément de plus et son dernier élément est l'élément ajouté.
- si on enlève (`dequeue`) un élément d'une file d'attente supposée non vide, la file contient alors un élément de moins et on retourne l'élément en tête de file.
- `isFull` détermine bien si la file d'attente est pleine.
- `isEmpty` détermine bien si la file d'attente est vide.

Questions

1. Traduire ces exigences en propriétés de la forme *assume*/*assert*, et les programmer en utilisant jUnit 5. Chaque méthode de test prendra la file d'attente à tester en paramètre.
2. *Tests structurels* : proposer des données de test pour ces primitives permettant d'obtenir la couverture des instructions, puis des décisions. On utilisera pour cela des tests paramétrés et une méthode auxiliaire `initQueue` permettant de créer une file d'attente déjà partiellement remplie (et de capacité égale à 16).
3. *Tests fonctionnels* : compléter vos tests avec des tests aléatoires. On utilisera cette fois une fonction auxiliaire `randomQueue` qui crée une file d'attente de taille aléatoire déjà partiellement remplie par des valeurs aléatoires (mais toujours de capacité fixe égale à 16). Une suite de 20 files d'attente aléatoires sera générée.